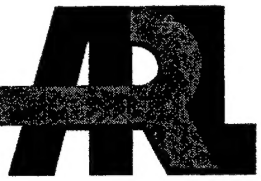
ARMY RESEARCH LABORATORY



Enabling Programmer-Controlled Combined Memory Consistency for Parallel Code Optimization

by Dixie M. Hisley

ARL-TR-3006 July 2003

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

Army Research Laboratory
Aberdeen Proving Ground, MD 21005-5067

ARL-TR-3006 **July 2003**

Enabling Programmer-Controlled Combined Memory Consistency for Parallel Code Optimization

Dixie M. Hisley
Computational and Information Sciences Directorate, ARL

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
July 2003	Final	October 2002 - March 2003

4. TITLE AND SUBTITLE
Enabling Programmer-Controlled Combined Memory Consistency for Parallel Code Optimization

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Dixie M. Hisley

5d. PROJECT NUMBER
BCHA06-3U21CL

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
U.S. Army Research Laboratory
ATTN: AMSRL-CI-HA
Aberdeen Proving Ground, MD 21005-5067

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-3006

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
In this report, a novel software technique is presented that provides the capability to guarantee a different memory consistency model for different variables within the same shared memory parallel program. Programmers need only focus on their parallel algorithm and which variables can tolerate the use of old values, while the technique automatically guarantees sequential consistency for all remaining variables and identifies where unnecessary synchronization overhead can be avoided. Program slicing is exploited to address the interactions between variables that can tolerate old values and those that require the most recent value to be read.

15. SUBJECT TERMS
software techniques, shared memory parallel programming, memory consistency

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dixie M. Hisley
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL	24	19b. TELEPHONE NUMBER *(Include area code)* 410-278-9156

Contents

List of Figures

INTENTIONALLY LEFT BLANK.

1. Introduction

Despite the purported cost effectiveness of workstation clusters, scientists and engineers continue to run large-scale, computationally intensive applications on high-end multiprocessor architectures. Shared memory architectures are preferred due to the relative ease of the sequential-to-parallel transformation, increasing scalability of these architectures, and their fast interprocessor communication. Scientists and engineers have been taking on the challenge of converting their applications into explicitly parallel programs to obtain the coarse-grain parallelism they desire. They rely on the compiler to hopefully identify and exploit the uniprocessor performance gains for each node processor and also to maintain a "correct" program execution based on their expectations of the shared memory behavior similar to that of a uniprocessor undertaking concurrent execution of multiple tasks. Unfortunately, potential performance gains are typically sacrificed, with much of the degradation due to the memory consistency model and the compiler's interactions with the hardware memory consistency model. Although optimized sequential code can outperform unoptimized compiled code by as much as the performance difference between two generations of processor hardware (Sarkar, 1997), classical compiler analysis and optimizations (Aho et al., 1986; Muchnick, 1997) were developed in the context of sequential programs designed to run on uniprocessors. These techniques do not account for updates to shared variables in threads other than the thread being analyzed. Thus, these compiler optimizations are typically turned off in order to avoid generating a program that does not behave as the user expects, given the underlying hardware memory consistency model. Unfortunately, performance degradation by turning off optimizations can be significant.

In this report, a novel software technique is presented that provides the capability to guarantee a different memory consistency model for different variables within the same shared memory parallel program. The key insight is that programmers are cognizant of which variables can tolerate old values and which variables need to be accessed in a way that all processors always see the most recently stored value. The first set of variables is referred to as **nonsecure** and the second set as **secure**. The set of **secure** variables typically does not include the entire variable set of an application.

In this approach, programmers add simple **secure** and **nonsecure** annotations to shared variable declarations to indicate their expectations, which are automatically propagated to the statements manipulating shared variables. A relaxed concurrent static single assignment (SSA) form of the program (Lee and Padua, 2000) is constructed so existing compiler optimization techniques for explicitly parallel, shared memory programs can be applied without modification. Sequential consistency is enforced selectively by the placement of fence instructions to ensure that the most recent value of a **secure** variable is always read. In other cases, the default is location

consistency (LC), which is based on the partial order execution semantics of parallel programs (Gao and Sarkar, 1994). The selective enforcement of sequential or location consistency (LC) is complicated by the interaction between the manipulation of **secure** and **nonsecure** variables throughout the program. The concept of program slicing is applied for parallel programs to identify and address these interactions.

To date, this is the first software technique to enable multiple memory consistency models to be applied within the same program. Previous research advocated the use of a single model for a given compiler/architecture combination. Parallel programmers should have the option to specify whether they want the semantics of a particular consistency model for different parts of a program based on their knowledge of their algorithm and data structures. The benefit is relieving the programmer of understanding memory consistency models, limiting the overhead of sequential consistency to those parts of the program related to the programmer's expectations of sequential consistency, and tailoring memory consistency modeling to individual programs automatically. Allowing this flexibility could significantly improve performance at run time.

The remainder of the report is organized as follows: Section 2 provides background on characteristics of applications and their parallelization, shared memory architectures, and memory consistency issues. Section 3 illustrates the uniprocessor optimization problem within a parallel programming environment and overviews previous work on this problem. Section 4 presents the technique for a programmer-controlled memory consistency model. Section 5 summarizes conclusions and future work.

2. Background

2.1 Characteristics of Parallel Applications

This research focuses on the loop-level and single program, multiple data (SPMD) shared memory parallel programming paradigms specifically targeted by OpenMP. For many shared memory programs, eliminating races on shared data accesses is necessary to produce correct repeatable results. These programs use some form of implied or explicit synchronization to avoid data races. If a data race does occur, it is usually considered a programming error. However, some algorithms rely on intentional data races as a natural part of their algorithm. An example is the set of iterative algorithms based on chaotic relaxation such as the parallel successive-over-relaxation (SOR) method. These algorithms can tolerate unsynchronized memory operations.

Nondeterministic parallel programs with data races are the most challenging for compiler analysis and optimization. However, one approach is to use a weaker memory consistency model, like LC. This weaker model does not consider data races to be errors, but instead gives them different semantics that allow the same flexibility in reordering data accesses as for parallel

programs that are deterministic or nondeterministic data-race-free. In this work, consideration is given to techniques to handle both deterministic and nondeterministic explicitly parallel shared memory programs. In particular, this work is motivated by the fact that very large applications can have both **secure** and **nonsecure** variables used in ways that whole subroutines could obtain higher performance from the LC model while other parts of the program are guaranteed to abide by the sequential consistency model to reflect programmer expectations.

2.2 Shared Memory Architectures and Memory Consistency

Regardless of implementation, the term "shared memory architecture" refers to a parallel computer where the address space is shared among multiple processors, and multiple processors are allowed to read and write the same memory locations. For correctness, a memory model is required that specifies the semantics of these potentially simultaneous memory operations. Sequential consistency is the simplest and most intuitive model to programmers. Many current multiprocessors support relaxed consistency models in hardware (Gharachorloo, 1995). However, these models may not be enough to guarantee correct execution of explicitly parallel programs for some hardware optimizations. Thus, these optimizations are typically turned off under relaxed consistency models so as to guarantee correct execution to explicitly parallel programs. In Lee's (1999) dissertation, he proposes to perform these optimizations at compile time. His proposed optimizing compiler does this by providing a sequentially consistent view of the underlying relaxed memory consistency model to programmers. He guarantees correct optimization under data races and the relaxed memory consistency model, while providing room for hardware-level optimizations.

It has been argued (Gao and Sarkar, 1994) that memory consistency models are too rigidly constrained by the memory coherence assumption and that this assumption should be discarded. Furthermore, an end-to-end view of memory consistency that can be understood at all levels of software and hardware is needed. As an alternative to sequential consistency and related memory models, an LC model has been proposed for this purpose. Instead of assuming that all writes to the same memory location are serialized according to some total order, the state of a memory location is modeled as a partially ordered multiset (pomset) of write and synchronization operations. There is no requirement for all processors to observe the same ordering of concurrent write operations.

As a result, the LC model provides a simple contract between the programmer (or compiler) and the hardware. Memory operations need not be serialized, only the partial order (the order of writes on a single processor with respect to that processor) defined by the program must be preserved. The LC model should be easier to implement because it relaxes the constraints imposed by sequential consistency models, and more performance optimizations should be possible. Finally, the LC model is applicable to asynchronous concurrent programs (e.g., SOR) that have intentional data races. Although no published implementations of the LC model in a

compiler have been reported, ongoing research is being conducted for the Java language using the Jalapeno research compiler infrastructure (Gao and Castelo, 2000).

3. Uniprocessor Optimization of Shared Memory Programs

Midkiff and Padua (1990) demonstrated that straightforward application of sequential optimization techniques within compilers for explicitly parallel shared memory programming fail to maintain correctness. Data races and synchronization issues make it impossible to apply classical methods directly to explicitly parallel programs.

As an example of an incorrect application of a classical optimization technique, consider the OpenMP program in Figure 1 when executed on two processors. In the code, the PARALLEL SECTIONS construct is a noniterative work-sharing construct that specifies that the enclosed sections of code are to be divided among the threads in the team. Each section is executed once by a thread in the team. The SHARED construct indicates that the variables listed on the parallel section are globally available to all threads for the duration of the enclosing parallel construct. The value of variable B at T = T + B should be 7 due to the assignment B = 7 by the second processor and the busy-waiting synchronization implied by the *do while* loops. If classical constant propagation is applied without considering the parallel execution of the two sections of code, variable B in the assignment T = T + B gets the value 9, due to the initial definition of B in processor 1's parallel section and no intervening redefinition before its use in the same section of code. Thus, by ignoring interactions between concurrent sections, the value 9 is incorrectly propagated to the use of B in T = T + B.

```
T = 0, B = 0, C = 0, D = 0                    // initialization
!$OMP  PARALLEL SECTIONS
                                              //  processor 1 executes this code
!$OMP& SHARED (T, B, C, D)                    //  variables shared by proc 1 and 2
B = 9                                                 // initial definition  of  b
C = 4
do while (D == 0)
end do
T = T + B                                     // use of  b, end of processor 1 work

!$OMP SECTION
                                              //  processor 2 executes this code once
                                              //  (in parallel with processor 1 work)
do while (C == 0)
end do
B = 7                                                 // redefinition of b
D = 1
                                              // end of processor 2 work
!$OMP END PARALLEL SECTIONS
```

Figure 1. Incorrect constant propagation.

In order to incorporate uniprocessor optimizations into a compiler for explicitly parallel programs, one must design an appropriate intermediate program representation and algorithms for analyzing when it is both safe and profitable to apply the various desired uniprocessor optimizations in the parallel program. The intermediate representation plays an important role in optimization, as it determines the ease with which analysis is performed and correct optimizations are identified. One intermediate representation that has been shown to enable efficient data flow analysis in sequential programs is the combination of a control flow graph (CFG) (Aho et al., 1986) with the SSA framework (Cytron et al., 1991).

Compilation techniques for explicitly parallel programs have not been aggressively researched. Some of the earliest papers on the analysis and optimization of explicitly parallel programs on shared memory computers explored the minimal set of delays that enforce sequentially consistent execution of concurrent processes (Krishnamurthy and Yelick, 1996; Midkiff et al., 1990; Shasha and Snir, 1988). In order to extend optimizations such as constant propagation to work correctly in a parallel programming environment, concurrent forms of the control flow graph (CFG) and the SSA framework were proposed by previous researchers (Grunwald and Srinivasan, 1993; Lee, 1999; Sarkar, 1997; Srinivasan et al., 1993). In particular, intermediate representations for parallel programs include the following: the parallel program graph (Sarkar, 1997), the parallel flow graph (Grunwald and Srinivasan, 1993), and the parallel precedence graph with SSA form (Srinivasan, et al., 1993). Lee (1999), Lee and Padua (2000), and Lee, et al. (1999) developed a concurrent static single assignment form (CSSA) based on the concurrent control flow graph (CCFG). Novillo (2000) and Novillo et al. (1998) extended the concurrent CFG and the CSSA form for programs with lock/unlock synchronization. Knoop and Steffen (1999) introduced efficient and optimal bit-vector analyses for parallel programs using the CFG.

In summary, previous research for explicitly parallel codes has focused on developing correctness criteria, intermediate program representations, and extending and applying classical data flow analysis and optimization techniques. However, much of the analysis is restricted to a subset of parallel programs and does not deal with all types of explicit synchronization. In addition, there has been a lack of consensus on an acceptable memory consistency model for developing correctness criteria (Gao and Sarkar, 1994). Furthermore, very few realistic implementations of the analysis and optimizations have been constructed.

4. Programmer-Controlled Memory Consistency

The overall goal is to give the programmer control and flexibility in memory consistency modeling via simple program annotations and to develop automatic techniques that translate the programmer's requests into modifications to the intermediate program representation in such a way that optimization and analysis algorithms require few changes. In this report, an approach

to programmer-controlled memory consistency is presented that is based on the data flow of the program.

In this approach, shared variables that need to satisfy the memory coherence assumption are annotated by the programmer as **secure**. Shared variables that do not need to satisfy the memory coherence assumption should be annotated as or be indicated by default to be **nonsecure**. The annotations are placed on the shared variable declaration statements indicated by the !$OMP SHARED construct (Figure 2). The scope of the annotation is the same as the scope of the shared variable (i.e., the parallel region associated with the parallel directive).

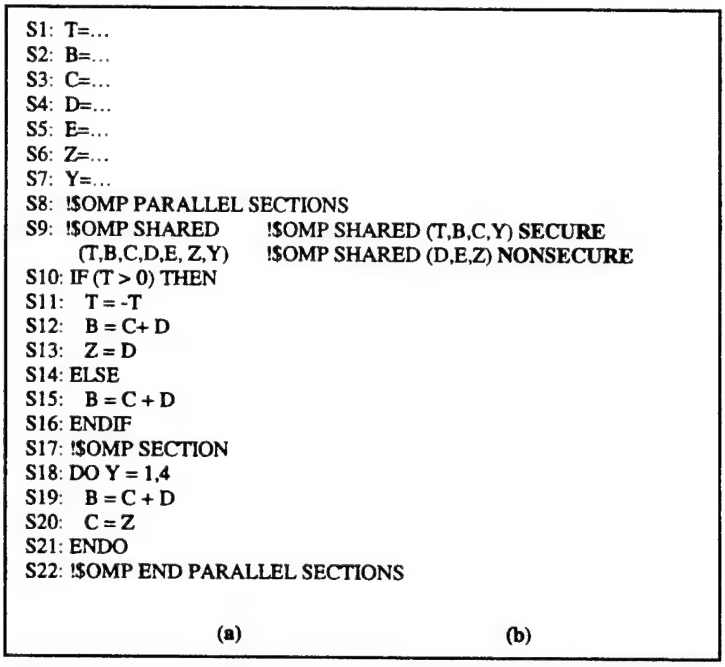


Figure 2. (a) example OpenMP (b) annotated Stmts.

4.1 Definitions

Sequential consistency has been historically defined in terms of the total program order for a sequential program or partial orderings for parallel programs. For this research, the definition of sequential consistency is extended for parallel programs to be defined in terms of a single variable with respect to a given point in a program. The definitions of sequentially consistent, **secure**, and **nonsecure** shared variables are given as follows:

- Definition 4.1, Sequentially Consistent Shared Variable—A sequentially consistent shared variable with respect to a given point P in a program requires that all updates to the shared variable that could affect the value of the variable at point P be immediately visible to other threads (i.e., abide by the memory coherence assumption).
- Definition 4.2, **Secure** Variable—A **secure** variable is a shared variable that is referenced by different threads that might execute concurrently in an explicitly parallel program and

needs to adhere to the memory coherence assumption at all program points P, and is therefore accessed according to the constraints imposed by sequential consistency.

- Definition 4.3, **Nonsecure** Variable—A **nonsecure** variable is a shared variable that is referenced by different threads that might execute concurrently in an explicitly parallel program and does not need to adhere to the memory coherence assumption, and therefore can be accessed according to the constraint imposed by LC (i.e., that each processor sequentially executes its node program).

Because the LC memory consistency model is used as the default model, **secure** variables that are involved in data races must be made to satisfy the memory coherence assumption. This can be accomplished by inserting constructs (by the programmer or by the compiler) to synchronize shared variable references or by the compiler automatically switching to a sequential consistency memory model to guarantee correctness. In this research, the approach is taken of automatically switching to a sequential consistency memory model within the software. Therefore, all shared variables that are annotated to be **secure** will be guaranteed to be sequentially consistent shared variables. All shared variables that are annotated to be **nonsecure** can default to LC whenever possible.

4.2 Base Intermediate Representation

The pioneering research of Lee (1999), Lee and Padua (2000), Lee et al. (1999), and Novillo (2000) and Novillo et al. (1998) has served as a starting point for this research. Their intermediate representations summarize control flow and information about the interaction between threads in a parallel program, thus enabling further exploration of explicitly parallel shared memory program optimization and analysis.

In particular, this approach is based on the CCFG program representation (Lee, 1999; Lee et al., 1999). The CCFG is a directed graph $G = \langle N, E, Entry_G, Exit_G \rangle$ such that N is the set of nodes in the graph (with each node corresponding to a basic block), E is the set of control flow edges, conflict edges, and synchronization edges, and $Entry_G$, $Exit_G$ are the unique entry and exit points of the program. A conflict edge is a bidirectional edge in the CCFG that joins any two basic blocks that can be executed concurrently (i.e., are located in separate threads of a *parallel section* or *parallel do*), reference the same shared variable, and one of the references is a write reference. There are two kinds of conflict edges: def-use and def-def.

A separate CCFG is generated for each procedure. Thus, a basic form of interprocedural analysis information can be gathered. At each procedure call, shared variables referenced and mutex bodies defined by the called procedure are propagated to the call site. This allows conflict and synchronization analysis to treat function calls almost as if they were inlined (Lee, 1999). Figure 3 presents an example CCFG with conflict edges for the sample code shown in Figure 2a. The one thread in this example contains an if-then-else construct, while the other thread contains a sequential *do loop*. The solid edges in the CCFG are control flow edges, while

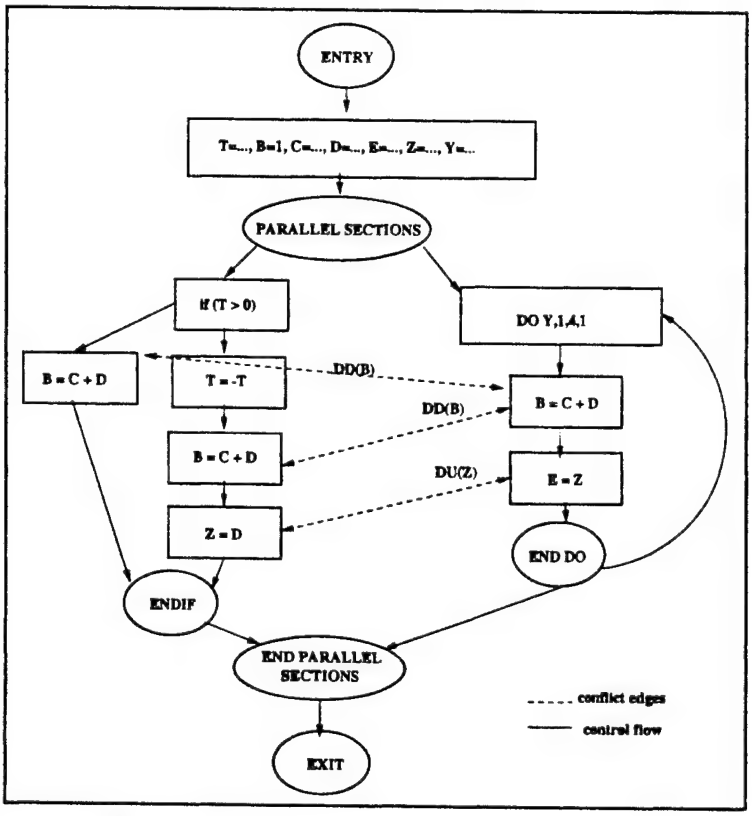


Figure 3. Example CCFG with conflicting edges.

conflict edges are represented by dashed edges. For example, there is a def-def edge linking the two assignments to variable B in the two different threads. A def-use edge for variable Z indicates the def-use relationship between the two threads created by the statements involving variable Z.

Initially, a CCFG representation of a program, which includes all conflict edges that would have to be enforced assuming a sequential consistency model for the whole program, is used. The software technique produces a relaxed CCFG in which conflict edges have been removed according to the programmer's annotations on shared variables.

4.3 Modeling Interactions of Variables

In order to ensure that all reads of a given **secure** variable indeed see the most recent value according to sequential memory consistency, sequential consistency is enforced for any value on which these reads depend, either directly or indirectly. In terms of the CCFG, the programmer is presented with sequential consistency, and the underlying relaxed consistency model of the hardware is hidden by inserting def-def and def-use conflict edges for each shared variable def-def or def-use dependency between different threads and by inserting fence instructions to ensure that the conflict edges are enforced.

In general, both def-def and def-use conflict edges for shared variables annotated as **nonsecure** could be removed and still maintain sequential consistency for all **secure** variables, as long as

nonsecure variables were manipulated completely in isolation of **secure** variables. However, programmers create dependencies among **secure** and **nonsecure** variables, which prevent the compiler from blindly removing all conflict edges for **nonsecure** variables.

The dependencies among reads and writes of **secure** and **nonsecure** variables can be specified in terms of four relations where LHS and RHS represent the left-hand side and right-hand side of a program statement, respectively.

(1) S: $LHS_{secure} = f(RHS_0, RHS_1,...)$, where LHS_{secure} is the definition of a **secure** variable as a function of a set of variables.

   (a) Each **secure** RHS variable in S represents a use of a **secure** variable within a statement defining another **secure** variable, LHS_{secure}. Any conflict edges leading into S due to these uses must be maintained for two reasons: (1) to ensure that the read of the RHS variable in S is the most recent value and (2) to ensure that the def of the **secure** variable in the LHS also maintains sequential consistency for later reads of LHS_{secure}.

   (b) Each **nonsecure** RHS variable in S represents a use of a **nonsecure** variable within a statement defining a **secure** variable. This case represents one way in which the manipulation of **secure** and **nonsecure** variables can interact. This case is discussed in more depth below.

(2) T: $LHS_{nonsecure} = f(RHS_0, RHS_1,...)$, where $LHS_{nonsecure}$ is the definition of a **nonsecure** variable as a function of a set of variables.

   (a) Each **nonsecure** RHS variable in T represents a use of a **nonsecure** variable $LHS_{nonsecure}$. Because we default to the LC model for **nonsecure** variables, any conflict edges leading into T due to the **nonsecure** RHS variable can be removed as long as the value of $LHS_{nonsecure}$ is not used later in a computation defining a **secure** variable. The def of the **nonsecure** RHS variable in this statement need not be performed prior to this read of the RHS variable.

   (b) Each **secure** RHS variable in T represents a use of a **secure** variable within a statement defining a **nonsecure** variable. This is the second case where **secure** and **nonsecure** variable manipulations interact.

In case 1(b), sequential consistency for LHS_{secure} implies that for all variable uses, RHS_i (in the computation of LHS_{secure}), all conflict edges leading to statement S for RHS_i, must be maintained in order to ensure that LHS_{secure} is kept **secure** for future reads. Thus, any RHS_i, that is **nonsecure** (call it $RHS_{nonsecure}$) must be treated as a **secure** variable for this particular use at S. Due to the requirement to treat $RHS_{nonsecure}$ as **secure** at this statement, conflict edges leading into S for $RHS_{nonsecure}$ must be maintained, and furthermore, any reaching def of $RHS_{nonsecure}$ at S must also be analyzed for dependencies on **nonsecure** variables.

For case 2(b), because the LHS is **nonsecure**, LC will be used for its def, and thus, typically any conflict edges leading into this statement for any of its RHS variables (which this statement reads) can be safely removed to reflect the more relaxed consistency model. However, the programmer has indicated that the most recent value of **secure** variables be read. A **secure** RHS variable in T would correspond to one of these reads. Thus, the conflict edges leading into T for any **secure** variables in the RHS must remain, regardless of the **nonsecure** status of the LHS.

The programmer annotated the shared variable declarations, but the dependencies between **secure** and **nonsecure** variables suggests that the compiler needs to determine additional locations where **nonsecure** variables need to be treated like **secure** variables in order to determine which conflict edges can be removed. This does not imply that all occurrences of the **nonsecure** variable need to be treated like a **secure** variable, only those manipulations on which a **secure** variable depends, either directly or indirectly. This leads to the following question:

How do we extend current correctness criteria, intermediate program representations, and optimization algorithms to handle both **secure** and **nonsecure** variables and their interactions and implementation as different memory consistency models simultaneously?

The key insight is that the relevant dependencies that need to be uncovered in order to implement different consistency models in different parts of a program based on data flow can be identified by program slicing. A program slice detects the parts of a program that statically may affect the values computed at some point (Weiser, 1984). A slice on a particular variable at a given program point will reveal the dependencies between **secure** and **nonsecure** variables on which this variable's value depends. For example, by slicing on an RHS variable at statement S, we can relabel its use at S and the variable defs and uses at statements that directly or indirectly affect it to be marked as **secure** when the LHS variable must be **secure** at S. Thus, for case 1(a), we need to slice on each **nonsecure** RHS, relabeling the RHS use as well as any variable manipulations that potentially affect it to be **secure**.

An algorithm for the static slicing of threaded programs based on CFGs and program dependence graphs has been developed by Krinke (1998). This algorithm is leveraged for use in this work. The input required for this slicing algorithm is a threaded program dependence graph (tPDG) and the slicing criterion (i.e., a node of the tPDG and the variable[s] to slice on). The output of the algorithm is the slice, which consists of a set of nodes of the tPDG.

The tPDG is similar to a CCFG; however, it includes control or parallel flow edges, direct control dependence edges, data dependence edges, and interference edges. The tPDG can be derived from the CCFG by the addition of the direct control and data dependence edges. Interference edges are identical to conflict edges. The slicing algorithm is referred to as function "parallel_slice" in Figure 4, which presents the pseudocode for our algorithm to implement programmer-controlled memory consistency.

Algorithm Construct_Relaxed_CCFG.

Input: Sequentially consistent CCFG G = ⟨N, E, $Entry_G$, $Exit_G$⟩ that includes all conflict edges as if all variables are **secure** in CSSA form, with added control and data dependence edges for slicing.

Output: A CCFG in CSSA form with conflict edges eliminated for **nonsecure** variables that can be handled using location consistency.

```
1: // Initialize flags on every def or use of shared variables
2: foreach shared variable S_i do
3: Initialize annotate_flag to secure or nonsecure to reflect annotation
4: end for
5: foreach shared variable S_i in node n_i ∈ N do
6: label_flag = annotate_flag
7: end for
8: // Perform backwards slicing to relabel nonsecure uses and defs*/
9:  foreach node n_i ∈ N with LHS_i label_flag = secure do
10:       foreach RHS_j with label_flag = nonsecure do
11:       // return slice w, a worklist containing nodes that affect RHS_i
12:         worklist w = parallel_slice (RHS_i, n_i)
13:         foreach node n_j ∈ w do
14:            foreach shared variable S_i in node n_j do
15:                label_flag = secure
16:            end for
17:         end for
18:      endfor
19: endfor
20: /* Test for case 2a and eliminate conflict edges
21:  foreach node n_i ∈ N do
22:       if LHS S_i label_flag = nonsecure then
23:       delete all def-def conflict edges to LHS variable of S_i
24:       delete all def-use conflict edges to nonsecure RHS variables of S_i
25:       endif
26: endfor
```

Figure 4. Algorithm for constructing relaxed CCFG.

4.4 Algorithm for Building Relaxed CCFG

There are three phases in the construction of a relaxed CCFG. First, an initialization phase is performed to build a sequentially consistent CCFG in CSSA form (which includes all conflict edges as if all variables are **secure**), extended to include control and data dependence edges. The algorithm takes this extended CCFG representation as input. In the algorithm, it is assumed that each node of the CCFG is an individual statement. Because **nonsecure** variables can be determined to require **secure** treatment at selected statements, the **secure** and **nonsecure** annotations on OpenMP shared variable declarations are propagated as flags on every def and use of the shared variables in the CCFG. In this way, programmers need only add annotations on variable declarations, but their expectations are propagated automatically to all defs and uses of the shared variables.

In the second phase, backward slicing is performed on each case 1(b) in the CCFG. During the backward slicing, any **nonsecure** variable reference found in the slice, including the original RHS variable, is relabeled to be **secure**. No conflict edges are removed during this phase.

Finally, the third phase eliminates conflict edges by a single pass over the CCFG using the final **secure** and **nonsecure** flags to identify all case 2(a) instances. Figure 4 presents pseudocode for our algorithm, which results in a relaxed CCFG.

In Figure 5, it is assumed that variables T, B, C, and Y are initially annotated by the programmer as **secure**, while variables D, E, and Z are annotated as **nonsecure** (as indicated in Figure 3b). The implications of these annotations are the removal of the def-use (Z) conflict edge and the relabeling of variable D to be **secure** in the two statements with C + D expressions because B is a **secure** variable being defined at those statements (case 1[b]). The algorithm for programmer-controlled memory consistency would slice from the statements B = C + D on variable D and relabel all **nonsecure** references in the program slice for D to become **secure**, in response to the **secure** annotation on B. Because variable Z is annotated to be **nonsecure**, variable Z in the statement Z = D can be location consistent, and the conflict edge to this LHS variable can be eliminated (case 2[b]).

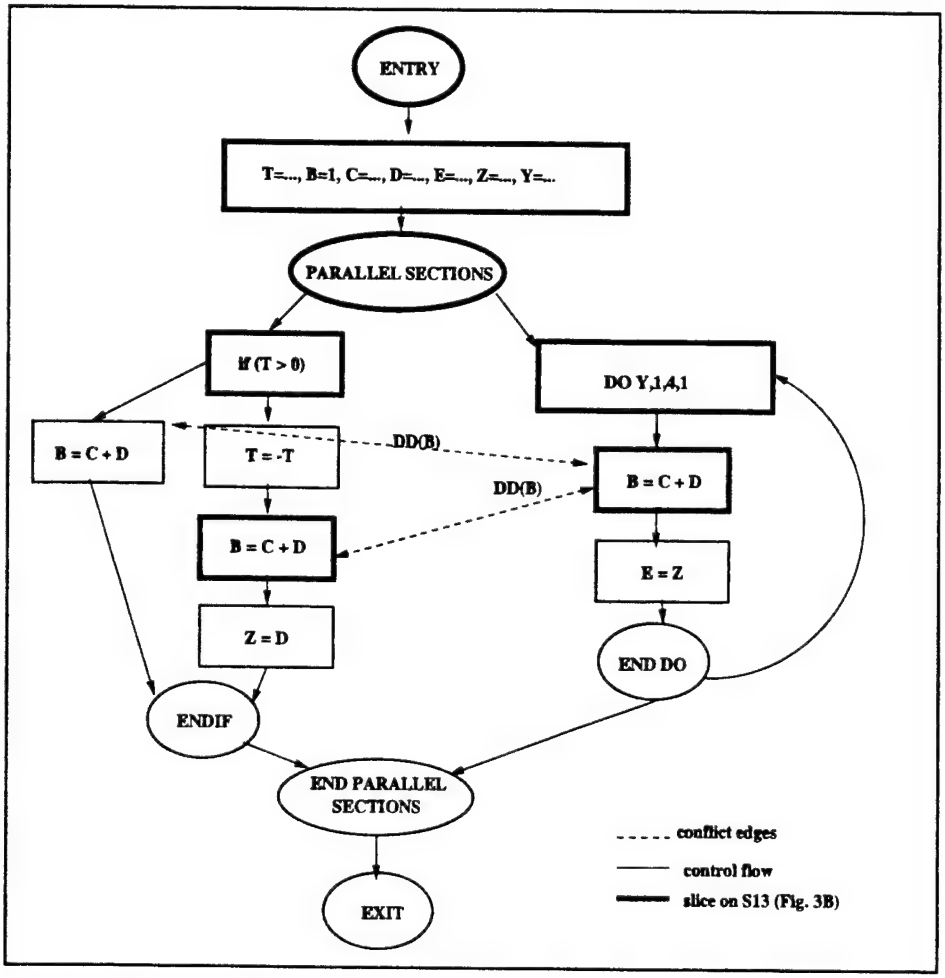


Figure 5. Relaxed CCFG with slicing.

The CCFG is built using a slightly modified version of a standard algorithm to build CFGs (Aho et al., 1986). The CSSAME form of the CCFG can be built in $O(r^3)$ time, where r is the maximum of the number of nodes, number of control edges, number of assignments, and number of variable references in the program (Novillo, 2000). The slicing algorithm takes the same time and space complexity as unthreaded slicing, with additional time complexity that could be exponential in the number of conflict edges in the worst case; however, much more reasonable times are expected for real programs given the characteristics of conflict edges that we have observed and that Krinke (1998) has observed. Slicing is performed once for each variable use labeled **nonsecure** in each statement with an LHS variable that is originally labeled as **secure**. Slicing is not repeatedly applied as relabeling occurs. The pass that eliminates the conflict edges requires O(N) time, where N is the number of nodes in the CCFG.

5. Conclusions and Future Work

The major contribution of this research is a technique for programmer-controlled memory consistency for explicitly parallel programs in which different models of consistency prevail in different parts of the program. Sequential consistency overhead is limited to where programmers want to ensure that most recent values are seen, while programmers are relieved of having to understand memory consistency models. Compiler optimizations can be tailored to different memory consistency models for different programs and for different parts of the program.

A software analysis tool is being designed based on this approach to perform program analysis and optimization for real scientific OpenMP programs (OpenMP Standard Board, 1997). An existing research compiler infrastructure, Odyssey, is being extended to incorporate the technique described in this research. Odyssey is an optimizing compiler for explicitly parallel programs with mutual exclusion synchronization incorporated into its data flow framework (Novillo, 2000). A set of scientific OpenMP benchmarks and sample programs have been gathered for testing. Among the codes most similar to real applications are *irreg*, *jacobi*, and *md*. *Irreg* simulates an unstructured computational fluid dynamics code, *jacobi* uses a SOR iterative algorithm to discretize the Helmholtz equation, and *md* simulates a molecular dynamics program. Experimentation, data collection, and analysis will be performed to determine improved performance in terms of cost, precision, and/or scalability in response to the new analysis and optimization techniques for OpenMP. Numerical output from the optimized and unoptimized programs will be compared for accuracy and precision. Metrics to be collected and/or computed include the following: optimized/unoptimized run-times, relative speedup, number of conflict edges eliminated, number of optimizations performed, and optimized/unoptimized size of the CCFG in nodes.

The approach to programmer-controlled memory consistency presented in this research is based on annotating shared variables and using data flow to partition the program's consistency

modeling. An alternate way of providing programmer-controlled memory consistency with the goal of allowing multiple models to be in effect during optimization based on the program and programmer's expectations using control structures is also a possibility for future investigation.

6. References

Aho, A.; Sethi, R.; Ullman, J. *Compilers: Principles, Techniques, and Tools;* Addison Wesley: NY, 1986.

Cytron, R.; Ferrante, J.; Rosen, B.; Wegman, M.; Zadek, F. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems* **October 1991**, *13* (4), 451–490.

Gao, G.; Sarkar, V. *Location Consistency: Stepping Beyond the Barriers of Memory Coherence and Serializability.* ACAPS Technical Memo 78; School of Computer Science, McGill University: Montreal, Quebec, 1994.

Gao, G.; Castelo, A. University of Delaware, Newark, DE. Private communication, 2000.

Gharachorloo, K. Memory Consistency Models for Shared Memory Multiprocessors. Ph.D., Stanford University, Stanford, CA, 1995.

Grunwald, D.; Srinivasan, H. Data Flow Equations for Explicitly Parallel Programs. *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Progr. (PPoPP'93)* **1993**, *28* (7), 159–168; ACM SIGPLAN Not.

Knoop, J.; Steffen, B. Code Motion for Explicitly Parallel Programs. *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Progr. (PPoPP '99)*, 1999, pp 13–24.

Krinke, J. Static Slicing of Threaded Programs. *Proceedings of the ACM SIGPLAN Workshop on Program Analysis for Software Tools and Engineering*, Montreal, CA, June 1998.

Krishnamurthy, A.; Yelick, K. Analyses and Optimizations for Shared Address Space Programs. *Parallel and Distributed Computing* **1996**, *38*, 139–144.

Lee, J. Compilation Techniques for Explicitly Parallel Programs. Ph.D., Department of Computer Science, University of Illinois at Urbana-Champaign, Chicago, IL, 1999.

Lee, J.; Padua, D. Hiding Relaxed Memory Consistency With Compilers. *Proceedings of International Conference on Parallel Architectures and Compilation Techniques,* 2000, 111–122.

Lee, J.; Padua, D.; Midkiff, S. Basic Compiler Algorithms for Parallel Programs. *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Progr. (PPoPP '99),* 1999.

Midkiff, S.; Padua, D. Issues in the Optimization of Parallel Programs. *Proceedings of the International Conference on Parallel Processing* **1990**, *II*, 105–113.

Midkiff, S.; Padua, D.; Cytron, R. Compiling Programs With User Parallelism. *Languages and Compilers for Parallel Computing* **1990**, 402–422.

Muchnick, S. *Advanced Compiler Design and Implementation;* Morgan Kaufmann: San Francisco, CA, 1997.

Novillo, D. Analysis and Optimization of Explicitly Parallel Programs. Ph.D., Department of Computing Science, University of Alberta, Edmonton, Canada, 2000.

Novillo, D.; Unrau, R.; Schaeffer, J. Concurrent SSA Form in the Presence of Mutual Exclusion. *Proceedings of the 1998 International Conference on Parallel Processing,* August 1998.

OpenMP Standard Board. *OpenMP Fortran Application Program Interface,* October 1997, Version 1.0, http://www.openmp.org.

Sarkar, V. Analysis and Optimization of Explicitly Parallel Programs Using the Parallel Program Dependence Graph Representation. *Lecture Notes in Computer Science* **1997**, *1366,* 94–113.

Srinivasan, H.; Hook, J.; Wolfe, M. Static Single Assignment for Explicitly Parallel Programs. *Proceedings of the 20th ACM Symposium on Principles of Programming Languages (POPL),* January 1993, 260–272.

Shasha, D.; Snir, M. Efficient and Correct Execution of Parallel Programs That Share Memory. *ACM Trans. Prog. Lang. Syst* **1988**, *10* (2), 282–312.

Weiser, M. Program Slicing. *IEEE Transactions on Software Engineering* **1984**, *10*, 352–357.

Bibliography

Bacon, D.; Graham, S.; Sharp, O. *Compiler Transformations for High-Performance Computing;* UCB/CSD-93-781; Computer Science Division, University of California: Berkeley, CA, 1993.

The CSSAME library for SUIFv1.0. http://www.cs.ualberta.ca/~jonathan/CSSAME/ (accessed 2001)

Ferrante, J.; Ottenstein, K.; Warren, J. The Program Dependence Graph and its use in Optimization. *ACM Transactions on Programming Languages and Systems* **July 1987**, *9* (3), 319–349.

Gharachorloo, K.; Lenoski, D.; Laudon, J.; Gibbons, P.; Gupta, A.; Hennessy, J. Memory Consistency and Event Ordering in Scalable Shared Memory Multiprocessors. *Proceedings of the 17th Annual International Symposium on Computer Architecture,* Seattle, WA, May 1990; pp 15–26.

Hisley, D.; Satya-Narayana, P.; Agrawal, G.; Pollock, L. Porting and Performance Evaluation of Irregular Codes Using OpenMP. *Concurrency Practice and Experience,* submitted for publication.

Hisley, D.; Satya-Narayana, P.; Agrawal, G.; Pollock, L. Porting and Performance of Irregular Codes Using OpenMP. *Proceedings of the 1st European Workshop on OpenMP (EWOMP),* Lund, Sweden, October 1999, pp 47–59.

Interim Java Grande Forum Report. http://www.javagrande.org/javagrande1999.html (accessed 2001)

Lamport, L. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers* **1979**, *28* (9), 690–691.

Morel, E.; Renvoise, C. Global Optimization by Suppression of Partial Redundancies. *Comm. ACM* **1979,** *22* (2), 96–103.

NAS Parallel Benchmarks Home Page. http://www.nas.nasa.gov/Software/NPB (accessed 2001)

Waheed, A.; Yan, J. *Parallelization of NAS Benchmarks for Shared Memory Multiprocessors;* NAS-98-010; March 1998.